Assignment

- Consider `x = y;`
- In Java, this makes `x` refer to whatever `y` refers to
  - `x` and `y` **share** the object
- In C, this **shallow-copies** `y` to `x`
  - if `x` & `y` are numbers, they're copied
  - if `x` & `y` are pointers, then the pointer is copied, but not what's pointed to
  - if `x` & `y` are structs, then the whole struct is copied, but not anything pointed to by that struct

An example

```
List list1;
List list2;
… // a bunch of operations to build list1
list2 = list1;  // what does this do?
… // a bunch of ops to extend list1
// now what's the state of list1? list2?
```


A variation

```
List* list1;
List* list2;
… // a bunch of operations to build list1
list2 = list1;  // what does this do?
… // a bunch of ops to extend list1
// now what's the state of list1? list2?
```


Tips

- Watch out for assignments doing (partial) copies behind your back
  - Using pointers to non-trivial data structures avoids this problem
- It's good to define your own (deep) copy functions that copy exactly what you want copied to duplicate the *abstract* state of your data structure

Arrays

- Key differences from Java arrays:
  - Created with a fixed length, cannot change
  - Length is not stored as part of array
    - No bounds checking
  - Arrays and pointers interchangeable

Array declarations

- Allocating a new array
  ```
  int x[10];  // an array of 10 integers
  char* y[20];  // an array of 20 ptrs-to-chars
  ```
  - Must use constant for array size
  ```
  #define LEN 30
  double z[LEN];
  ```
- Use `a[i]` notation to read/write array elems
  ```
  x[i] = x[j] + 1;
  ```
- No length stored with array

Arrays in memory

- For a declaration of the form
  `type name[len];`
  memory is allocated to hold *`len`* copies of *`type`* values
  - No length field allocated
- *`name`* is a pointer to the first element

Arrays as pointers

- An array can be treated as a pointer to its first element

```
int  a[20];
int* b = a;     // works
int* c = &a[0];// same effect
```

- Look at memory layout to see why

Arrays in the heap

- Can allocate arrays in the heap using `new`
  - Returns a pointer to the first element

```
int* a = new int[20];
```

- Can deallocate like any pointer to heap

```
delete a;
```


Array function arguments

- Can pass an array to a function, or return an array
  - Actually, returning the pointer to the first element
- For arguments (but not results), can declare an array whose length is omitted

```
int* f(int a[]) {
  return a;
}
```

  - Allows arrays of different lengths to be passed to the function

Using argument arrays

- Q: If I get an array as an argument, how can I use it? How do I know how long it is?
- A: Must pass the length of the argument array as an extra argument

```
int x[20];
…
f(x, 20);
…
```

```
void f(int a[], int n) {
   for (int i = 0; i < n; i++) {
      a[i] = a[i] + a[n-i-1];
   }
}
```


Multidimensional arrays

- Can declare matrices/arrays with multiple dimensions
  - Like Java, they're declared & accessed as arrays of arrays of arrays of …
  - Unlike Java, one large memory block is allocated for the whole matrix
    - "row-major order"

Example

```
#define numRows 10
#define numCols 20
double m[numRows][numCols];
for (int r = 0; r < numRows; r++) {
  double* row = m[r];  // OK: ptr to rth row
  for (int c = 0; c < numCols; c++) {
     int elem = row[c];  // == m[r][c]
  }
}
```


Strings

- In Java, `String` is a library class, with lots of cool operations
  - Plus, special "…" syntax and + operation
- In C, a string is just an array of chars, ending in a `'\0'` (null) character
  - Similar "…" syntax, implicitly including `'\0'`
  - `#include <string.h>` to get lots of library functions that work over null-terminated arrays of characters, a.k.a. strings

Issues

- Like all arrays, no length stored in a string
  - Must search for null character to find length
  - Different than array length!
- Cannot store a null character in a string
  - Not suitable for binary data
  - Must guard in face of external input
- `char*` and `char[]` both suggest "string", but not necessarily

String operations

- Do "`man string`" to find out many string operations
  - Generally, less friendly than Java, due to lack of internal length and avoidance of allocation
- E.g.:
  - `int strlen(char* s);`
  - `int strcmp(char* s1, char* s2);`
  - `char* strdup(char* src);`
  - `char* strncpy(char* dest, char* src, int max);`

Casting

- C programs allow unrestricted casting from one type to another
  - Some casts are conversions
    - E.g., between different numeric types
  - Some casts restrict or reveal information
    - E.g. between pointers to structs with more or fewer fields
    - `void*` is the implicit "supertype" of all pointers, akin to `Object` in Java
  - Some casts just reinterpret the bits
    - E.g. between an int and a pointer

"Generic" code

- One use for casting to write one piece of code that's generic across many possible client types
- E.g., a `List` of things, where we don't want to restrict what kind of things we can store
  - In Java: use `Object` as "universal" type, cast arguments to `Object` (implicitly) when put in and cast back to real type (explicitly) when take out
    - Except that primitive types aren't `Objects` L
  - In C: `long`, or `void*`, or unions, or …
  - In C++: templates

Example

```
struct Link {
  void* data;
  Link* next;
};
Link* addFirst(Link* list, void* data) {
  … }
…
Link* myList = NULL;
myList = addFirst(myList, "a string");
char* firstElem = (char*) myList->data;
```


A taste of templates

```
template <class T> struct Link {
  T data;
  Link<T>* next;
};
template <class T>
 Link<T>* addFirst(Link<T>* list, T data)
  {…}
…
Link<const char*>* myList = NULL;
myList = addFirst(myList, "a string");
const char* firstElem = myList->data;
```


Input/output library functions

- `printf` has many ways of producing formatted output
  - `cout` is C++ alternative that many prefer
- `scanf` is way to get input from stdin
  - `cin` is C++ alternative
  - note: pass *pointers* as arguments
- look up `fopen`, `fread`, `fwrite`, `fclose` to do file I/O

More useful features

- "`const`" can be put before a type to make that thing read-only
  - E.g. "`const char*`" is a pointer to a character (or character array) that can be read but not modified
- Enums are a nice way to declare a bunch of named integer constants and a integral type
  - E.g.: `enum FlagColor {RED,WHITE,BLUE};`
- Refs (`&`) are an alternative to pointers (`*`) that are never null and that automatically dereference
  - Good for call-by-reference arguments